

---

# **Materials Modelling**

**Weixuan Zhang**

**Nov 26, 2020**



**CONTENTS:**

<b>1</b>	<b>Morse module</b>	<b>3</b>
<b>2</b>	<b>MorseFast module</b>	<b>5</b>
<b>3</b>	<b>UnitCell module</b>	<b>7</b>
<b>4</b>	<b>pairwise module</b>	<b>9</b>
<b>5</b>	<b>hydrostatic module</b>	<b>11</b>
<b>6</b>	<b>shear module</b>	<b>13</b>
<b>7</b>	<b>util module</b>	<b>15</b>
<b>8</b>	<b>Indices and tables</b>	<b>17</b>
	<b>Python Module Index</b>	<b>19</b>
	<b>Index</b>	<b>21</b>



Modules ==



## MORSE MODULE

```
class Morse.MorsePotential (**kwargs)
    Bases: ase.calculators.calculator.Calculator
```

```
calculate (atoms=None, properties=None, system_changes=['positions', 'numbers', 'cell', 'pbc', 'initial_charges', 'initial_magmoms'])
```

Do the calculation.

**properties: list of str** List of what needs to be calculated. Can be any combination of ‘energy’, ‘forces’, ‘stress’, ‘dipole’, ‘charges’, ‘magmom’ and ‘magmoms’.

**system\_changes: list of str** List of what has changed since last calculation. Can be any combination of these six: ‘positions’, ‘numbers’, ‘cell’, ‘pbc’, ‘initial\_charges’ and ‘initial\_magmoms’.

Subclasses need to implement this, but can ignore properties and system\_changes if they want. Calculated properties should be inserted into results dictionary like shown in this dummy example:

```
self.results = {'energy': 0.0,
                'forces': np.zeros((len(atoms), 3)),
                'stress': np.zeros(6),
                'dipole': np.zeros(3),
                'charges': np.zeros(len(atoms)),
                'magmom': 0.0,
                'magmoms': np.zeros(len(atoms))}
```

The subclass implementation should first call this implementation to set the atoms attribute and create any missing directories.

```
default_parameters: Dict[str, Any] = {'D': 0.16156, 'alpha': 2.0926, 'r0': 2.6163,
implemented_properties: List[str] = ['energy', 'forces', 'stress', 'local_energy']
morse_pair_energy (r)
morse_pair_energy_deriv (r)
nolabel = True
```





**MORSEFAST MODULE**



## UNITCELL MODULE

Calculations on a unit cell

**class** `UnitCell.CuCell` (*a: float = 3.6*)

Bases: `object`

A class for the Cu unit cell, with methods for applying strain and shear

**cu**

an object representing the atoms

**Type** `ase.Atoms`

**default\_a**

default unit cell size

**Type** `float`

**default\_a = 3.6**

**classmethod** `from_default_eq_strain` (*strain: float*) → `UnitCell.CuCell`

Create the class from strain (with respect to the default unit cell size)

The is used to create a unit cell with size where the potential energy is minimum, as calculated elsewhere.

**Parameters** **strain** (*float*) – the strain

**Returns** class created

**Return type** `CuCell`

**hydrostatic\_deform** (*strain: float*) → `ase.atoms.Atoms`

Function that returns the deformed unit cell under a given hydrostatic strain

**Parameters** **strain** (*float*) – the hydrostatic strain applied

**Returns** deformed unit cell

**Return type** `ase.Atoms`

**property** `init_cell`

Get initial cell vectors

**Returns** initial cell vectors (columns)

**Return type** `np.ndarray`

**property** `init_vol`

Get initial unit cell volume

**Returns** initial volume ( $\text{\AA}^3$ )

**Return type** `float`

**set\_cell** (*cell*: *numpy.ndarray*) → None

Set the unit cell length

**Parameters** **cell** (*float*) – the new unit cell

**shear\_deform** (*shear*: *float*) → ase.atoms.Atoms

Function that returns the deformed unit cell under a shear in Y direction

**Parameters** **shear** (*float*) – shear in Y direction

**Returns** deformed unit cell

**Return type** ase.Atoms

**strain\_deform** (*strain\_x*: *float*, *strain\_y*: *float*, *strain\_z*: *float*) → ase.atoms.Atoms

Function that returns the deformed unit cell under normal strains

**Parameters**

- **strain\_x** (*float*) – strain in the x direction
- **strain\_y** (*float*) – strain in the y direction
- **strain\_z** (*float*) – strain in the z direction

**Returns** deformed unit cell

**Return type** ase.Atoms

**visualize** () → None

Visualize the unit cell

## PAIRWISE MODULE

Calculations involving a pair of Cu atoms

`pairwise.build_pair` (*d0*: *Union[float, int] = 1*) → Callable

Closure to store the atoms object

**Parameters** *d0* (*Union[float, int]*, *optional*) – default unit cell length

**Returns** function to apply strain

**Return type** Callable

`pairwise.get_pair` (*d*: *Union[float, int]*) → ase.atoms.Atoms

Function that returns the deformed unit cell under a given hydrostatic strain

**Parameters** *d* (*Union[float, int]*) – distance (Å)

**Returns** deformed atom pair

**Return type** Atoms

`pairwise.get_pairwise_force` (*d*: *Union[float, int]*) → float

Calculate the force between two atoms separated by the given distance

**Parameters** *d* (*Union[float, int]*) – distance (Å)

**Returns** force (eV/Å)

**Return type** float

`pairwise.get_pairwise_forces` (*arr*: *numpy.ndarray*) → *numpy.ndarray*

Apply pairwise force calculation to an array of distances

**Parameters** *arr* (*np.ndarray*) – array of distances (Å)

**Returns** array of forces (eV/Å)

**Return type** *np.ndarray*

`pairwise.get_pairwise_pe` (*d*: *Union[float, int]*) → float

Calculate the potential energy of two atoms separated by the given distance

**Parameters** *d* (*Union[float, int]*) – distance (Å)

**Returns** potential energy (eV)

**Return type** float

`pairwise.get_pairwise_pes` (*arr*: *numpy.ndarray*) → *numpy.ndarray*

Apply pairwise potential energy calculation to an array of distances

**Parameters** *arr* (*np.ndarray*) – array of distances (Å)

**Returns** array of potential energies (eV)

**Return type** np.ndarray

## HYDROSTATIC MODULE

Calculations related to hydrostatic loading

`hydrostatic.cu_cell`

instance of CuCell class with default lattice vector size

**Type** *CuCell*

`hydrostatic.get_hydrostatic_pe(strain: float) → float`

Calculate the potential energy after applying a hydrostatic strain

**Parameters** `strain` (*float*) – strain

**Returns** potential energy (eV)

**Return type** *float*

`hydrostatic.get_hydrostatic_pes(arr: numpy.ndarray) → numpy.ndarray`

Apply the potential energy calculation to an array of strains

**Parameters** `arr` (*np.ndarray*) – array of strains

**Returns** array of potential energies (eV)

**Return type** *np.ndarray*

`hydrostatic.get_hydrostatic_pressure(strain: float) → float`

Calculate the pressure from the stress matrix

**Parameters** `strain` (*np.ndarray*) – strain

**Returns** hydrostatic pressure (eV/Å<sup>3</sup>)

**Return type** *float*

`hydrostatic.get_hydrostatic_pressures(arr: numpy.ndarray) → numpy.ndarray`

Apply the pressure calculation to an array of strains

**Parameters** `arr` (*np.ndarray*) – array of strains

**Returns** array of pressures (eV/Å<sup>3</sup>)

**Return type** *np.ndarray*

`hydrostatic.get_hydrostatic_stress(strain: float) → numpy.ndarray`

Calculate the stress after applying a hydrostatic strain

**Parameters** `strain` (*float*) – strain

**Returns** stress (eV/Å<sup>3</sup>)

**Return type** *float*

`hydrostatic.get_hydrostatic_vol(strain: float) → float`

Calculate the new volume after applying the strain

**Parameters** `strain` (*float*) – strain

**Returns** new volume ( $\text{\AA}^3$ )

**Return type** `float`

`hydrostatic.get_hydrostatic_vols(arr: numpy.ndarray) → numpy.ndarray`

Apply the deformed volume calculation to an array of strains

**Parameters** `arr` (*np.ndarray*) – array of strains

**Returns** array of volumes ( $\text{s\AA}^3$ )

**Return type** `np.ndarray`



## SHEAR MODULE

Calculations related to shear

`shear.cu_cell`

instance of CuCell class with default lattice vector size

**Type** *CuCell*

`shear.get_shear_stress(shear: float) → numpy.ndarray`

Calculate the stress after applying a shear in  $y$  direction

The shear tensor is

$$\begin{bmatrix} \sigma_x & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_y & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_z \end{bmatrix}$$

if the stress is only applied in the  $y$  direction,  $\tau_{xz} = \tau_{yz} = 0$

**Parameters** `shear` (*float*) – shear

**Returns** shear stress  $\tau_{xy}$  (eV/Å<sup>3</sup>)

**Return type** float



## UTIL MODULE

Helper functions

`util.map_func` (*func*: Callable[[Union[float, int]], float], *arr*: numpy.ndarray) → numpy.ndarray

Mapping a function over a Numpy array

**Parameters**

- **func** (Callable[[Union[float, int]], float]) – function applied to each element of the array
- **arr** (np.ndarray) – array to be mapped over

**Returns** transformed array

**Return type** np.ndarray

`util.x_of_min` (*x*: numpy.ndarray, *y*: numpy.ndarray) → Union[int, float]

Get the value of x where the y is minimum

**Parameters**

- **x** (np.ndarray) – array of x
- **y** (np.ndarray) – array of y

**Returns** x of the minimum point

**Return type** Union[int, float]



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### h

hydrostatic, [11](#)

### m

Morse, [3](#)

### p

pairwise, [9](#)

### s

shear, [13](#)

### u

UnitCell, [7](#)

util, [15](#)





## B

`build_pair()` (in module *pairwise*), 9

## C

`calculate()` (*Morse.MorsePotential* method), 3

`cu` (*UnitCell.CuCell* attribute), 7

`cu_cell` (in module *hydrostatic*), 11

`cu_cell` (in module *shear*), 13

`CuCell` (class in *UnitCell*), 7

## D

`default_a` (*UnitCell.CuCell* attribute), 7

`default_parameters` (*Morse.MorsePotential* attribute), 3

## F

`from_default_eq_strain()` (*UnitCell.CuCell* class method), 7

## G

`get_hydrostatic_pe()` (in module *hydrostatic*), 11

`get_hydrostatic_pes()` (in module *hydrostatic*), 11

`get_hydrostatic_pressure()` (in module *hydrostatic*), 11

`get_hydrostatic_pressures()` (in module *hydrostatic*), 11

`get_hydrostatic_stress()` (in module *hydrostatic*), 11

`get_hydrostatic_vol()` (in module *hydrostatic*), 11

`get_hydrostatic_vols()` (in module *hydrostatic*), 12

`get_pair()` (in module *pairwise*), 9

`get_pairwise_force()` (in module *pairwise*), 9

`get_pairwise_forces()` (in module *pairwise*), 9

`get_pairwise_pe()` (in module *pairwise*), 9

`get_pairwise_pes()` (in module *pairwise*), 9

`get_shear_stress()` (in module *shear*), 13

## H

`hydrostatic`

module, 11

`hydrostatic_deform()` (*UnitCell.CuCell* method), 7

## I

`implemented_properties` (*Morse.MorsePotential* attribute), 3

`init_cell()` (*UnitCell.CuCell* property), 7

`init_vol()` (*UnitCell.CuCell* property), 7

## M

`map_func()` (in module *util*), 15

module

*hydrostatic*, 11

*Morse*, 3

*pairwise*, 9

*shear*, 13

*UnitCell*, 7

*util*, 15

*Morse*

module, 3

`morse_pair_energy()` (*Morse.MorsePotential* method), 3

`morse_pair_energy_deriv()` (*Morse.MorsePotential* method), 3

*MorsePotential* (class in *Morse*), 3

## N

`nolabel` (*Morse.MorsePotential* attribute), 3

## P

*pairwise*

module, 9

## S

`set_cell()` (*UnitCell.CuCell* method), 7

*shear*

module, 13

`shear_deform()` (*UnitCell.CuCell* method), 8

`strain_deform()` (*UnitCell.CuCell* method), 8

### U

UnitCell  
    module, [7](#)  
util  
    module, [15](#)

### V

visualize() (*UnitCell.CuCell method*), [8](#)

### X

x\_of\_miny() (*in module util*), [15](#)